JavaScript - Ajax 4

async / await

Son instrucciones nuevas introducidas en ES2017 que permiten escribir el código de peticiones asíncronas como si fueran síncronas, lo que facilita su comprensión.

Hay que tener en cuenta que NO están soportadas por navegadores antiguos.

Si hubiéramos utilizado async/await en el primer ejemplo que hicimos, sí habría funcionado.

Se puede llamar a cualquier función asíncrona (por ejemplo, una promesa como *fetch*) anteponiendo la palabra **await** a la llamada. Esto provocará que la ejecución se "espere" a que se resuelva la promesa devuelta por esa función. Así nuestro código se asemeja a código síncrono ya que no continúan ejecutándose las instrucciones que hay después de un *await* hasta que esa petición se ha resuelto.

Cualquier función que realice un *await* pasa a ser asíncrona ya que no se ejecuta en ese momento, sino que se espera un tiempo.

Para indicarlo debemos anteponer la palabra **async** a la declaración *`function`*. Al hacerlo, automáticamente se "envuelve" esa función en una promesa (o sea, que esa función pasa a devolver una promesa, a la que podríamos ponerle un `await` o un `.then()`).

Siguiendo con el ejemplo anterior, el código Javascript sería:

```
async function pideDatos() {
    const response = await fetch('http://localhost:4000/productos?id=' + idProd);
    if (!response.ok) {
```

```
    throw `Error ${response.status} de la BBDD: ${response.statusText}`
  }
  const myData = await response.json(); // recuerda que .json() tb es una promesa
  return myData;
}
...
// Y llamaremos a esa función con
const myData = await pideDatos();
```

Nota: solo se puede utilizar **await** dentro de funciones declaradas con la palabra reservada **async.**

Observa la diferencia: si hacemos:

```
const response = fetch('http://localhost:4000/productos?id=' + idProd);
```

obtenemos en *response* una promesa, y para obtener el valor se debería hacer `response.then()`. Pero si hacemos:

```
const response = await fetch('http://localhost:4000/productos?id=' + idProd);
```

lo que obtenemos en *response* es ya el valor devuelto por la promesa cuando se resuelve.

Con esto conseguimos que llamadas asíncronas se comporten como instrucciones síncronas, lo que aporta claridad al código.

Hay algunos ejemplos del uso de *async / await* en la página de MDN.

Siguiendo con el ejemplo de obtener datos de un producto indicando su id:

El código HTML sería igual que antes:

```
<form id="getProduct">
     <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
     <br>
    <button type="submit">Buscar</button>
    <p id="p1">Aquí vendrán los datos</p>
</form>
```

Y el código Javascript sería:

```
const SERVER = 'http://localhost:4000';

window.addEventListener('load', () => {
  document.getElementById('getProduct').addEventListener('submit', async (event) => {
    event.preventDefault();
    let idProd = document.getElementById('id-prod').value
    if (isNaN(idProd) || idProd.trim() == '') {
      alert('Debes introducir un número')
    } else {
      const datos = await getData(idProd)
      // La ejecución se para en la sentencia anterior hasta que
      // contesta la función getData
      document.getElementById('p1').innerHTML = datos[0].name+"  "+datos[0].descrip;
    }
  })
```

```
})

async function getData(idProd) {
  const response = await fetch(SERVER + '/productos?id=' + idProd)
  if (!response.ok) {
    throw `Error ${response.status} de la BBDD: ${response.statusText}`
  }
  const datos = await response.json()
  return datos
}
```

Gestión de errores en async/await

En el código anterior no estamos tratando los posibles errores que se pueden producir.

Con *async / await* los errores se tratan como en las excepciones, con *try ... catch*.

El código que resulta para el ejemplo de pintar los datos de un producto indicando su id:

El código HTML sería igual que siempre:

```
    <form id="getProduct">
         <label for="id-prod">Id Producto: </label><input type="number" name="id-prod" id="id-prod">
         <br>
        <button type="submit">Buscar</button>
        <p id="p1">Aquí vendrán los datos</p>
    </form>
```

Y el código Javascript sería:

```
// fetch con async/await
// tratando errores con try-catch
const SERVER = 'http://localhost:4000'

window.addEventListener('load', () => {
  document.getElementById('getProduct').addEventListener('submit', async (event) => {
    event.preventDefault();
    let idProd = document.getElementById('id-prod').value
    if (isNaN(idProd) || idProd.trim() == '') {
      alert('Debes introducir un número')
    } else {
        try {
            const datos = await getData(idProd)
            // La ejecución se para en la sentencia anterior hasta que
            // contesta la función getData
            document.getElementById('p1').innerHTML = datos.name+"  "+datos.descrip;
        } catch (err) {
            console.log("mal");
            console.error(err);
            return;
        }
    }
  })
})
```

```
async function getData(idProd) {
  const response = await fetch(SERVER + '/productos?id=' + idProd)
  if (!response.ok) {
    throw `Error ${response.status} de la BBDD: ${response.statusText}`
  }
  const datos = await response.json()
  return datos
}
```

También podemos tratar los errores sin usar *try…catch,* porque como una función asíncrona devuelve una promesa, podemos suscribirnos directamente a su *.catch*

Ejemplos de GET y POST con Fetch (utilizando y sin usar async-await)

Con nuestro fichero productos.json:

```
{
  "productos": [
    {
      "id": 1,
      "name": "Teclado",
      "descrip": "Teclado mecánico Cherry ps/2"
    },
    {
      "id": 2,
      "name": "Moninor Ph-21",
      "descrip": "Monitor Phillips SVGA 21 \""
    },
    {
      "id": 3,
      "name": "Ratón Logi-Laser",
      "descrip": "Tatón Láser Logitech Pro USB"
    }
  ]
}
```

Index.html para GET

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <form id="getProduct">
        <label for="id-prod">Id Producto: </label><input type="number" name="id-prod"
id="id-prod">
         <br>
```

```
        <button type="submit">Buscar</button>
        <p id="p1">Aquí vendrán los datos</p>
    </form>
    <script src="js/get.js"></script>

</body>
</html>
```

Código javascript

```
// GET con fetch
const SERVER= 'http://127.0.0.1:4000';
window.addEventListener("load", ()=>{
  document.getElementById('getProduct')
    .addEventListener('submit', (event)=>{
    event.preventDefault();
    let idProd=document.getElementById('id-prod').value;
    let promesa=fetch(SERVER + '/productos?id=' + idProd);
          //obtenemos una promesa
    promesa
      .then((dato1)=>dato1.json())
          //ejecuta lo de dentro, dato1 es lo que then ha
recibido
      .then((dato2)=>{  // dato2 es lo que recibe del
anterior then
        console.log(dato2);
        document.getElementById('p1').innerText=dato2[0].nam
e;
      })
      .catch((problema)=> document.getElementById('p1')
            .innerText="Ha habido error: "+problema);

  });
});
```

```
// fetch con async await
// se trata de separar la zona donde
// se pinta, de la petición a la API.
// ya no usamos then, porque await garantiza
// que ya llegó la respuesta
const SERVER= 'http://127.0.0.1:4000';
window.addEventListener("load", ()=>{
  document.getElementById('getProduct')
              .addEventListener('submit', async
(event)=>{
    event.preventDefault();
    const idProd=document.getElementById('id-
prod').value;
    const dato1=await getData(idProd);
    document.getElementById('p1').innerText=dato1[
0]
          .name + " " +dato1[0].descrip;
  });
});

async function getData(idProd){
  const dato1 = await
fetch(SERVER+'/productos?id='+idProd);
  // lo normal sería mirar si hubo error:
  /* if (!dato1.ok) {
    throw `error${dato1.status}
${dato1.statusText}`}
  */
  console.log(dato1);
  const dato2 = await dato1.json();
  console.log(dato2);
  return dato2;
}
```

```
// fetch con
const SERVER= 'http://127.0.0.1:4000';
window.addEventListener("load", ()=>{
  document.getElementById('getProduct').addEventListener('submit', (event)=>{
    event.preventDefault();
    let idProd=document.getElementById('id-prod').value;
    let promesa=fetch(SERVER + '/productos?id=' + idProd); //obtenemos una promesa
    promesa
      .then((dato1)=>dato1.json()) //ejecuta lo de dentro, dato1 es lo que then ha recibido
      .then((dato2)=>{  // dato2 es lo que recibe del anterior then
        console.log(dato2);
        document.getElementById('p1').innerText=dato2[0].name;
      })
      .catch((problema)=> document.getElementById('p1').innerText="Ha habido error: "+problema);

  });
});
```

```
// fetch con async await
// se trata de separar la zona donde se pinta, de la petición a la API.
// ya no usamos then, porque await garantiza que ya llegó la respuesta
const SERVER= 'http://127.0.0.1:4000';
window.addEventListener("load", ()=>{
  document.getElementById('getProduct').addEventListener('submit', async (event)=>{
    event.preventDefault();
    const idProd=document.getElementById('id-prod').value;
    const dato1=await getData(idProd);
    document.getElementById('p1').innerText=dato1[0].name + " " +dato1[0].descrip;
  });
});

async function getData(idProd){
  const dato1 = await fetch(SERVER + '/productos?id=' + idProd);
  // lo normal sería mirar si hubo error:
  // if (!dato1.ok) { throw `error${dato1.status} ${dato1.statusText}`}
  console.log(dato1);
  const dato2 = await dato1.json();
  console.log(dato2);
  return dato2;
}
```

Index.html para POST

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="shortcut icon" href="#" type="image/x-icon">
    <title>Document</title>
</head>
<body>
    <form id="addProduct">
        <label for="name">Nombre: </label><input type="text" name="name" id="name"
required><br>
        <label for="descrip">Descripción: </label><input type="text" name="descrip"
id="descrip" required><br>

        <button type="submit">Agregar Producto</button>
        <p id="p1"> ...</p>
    </form>
    <script src="js/post.js"></script>
</body>
</html>
```

Código JavaScript

```
// POST con fetch

const SERVER = 'http://localhost:4000';

window.addEventListener('load', ()=>{
  document.getElementById('addProduct').addEventListener('sub
mit', (event)=>{
    event.preventDefault();
    const nuevoProducto={
      id:"",
      name:document.getElementById('name').value,
      descrip:document.getElementById('descrip').value
    };
```

```
// POST con fetch y asunc await
const SERVER = 'http://localhost:4000';
async function anadirProducto(nuevoProducto){
  const dato1 = await fetch(SERVER+'/productos',
{
    method: 'POST',
    body: JSON.stringify(nuevoProducto),
    headers: {
      'Content-Type': 'application/json'
    }
  });
  const dato2= await dato1.json();
  return dato2;
```

```
    const promesa=fetch(SERVER+'/productos', {
      method: 'POST',
      body: JSON.stringify(nuevoProducto),
      headers: {
        'Content-Type': 'application/json'
      }
    })
    promesa
      .then((dato1)=>dato1.json())
      .then((dato2)=>{
        //pintamos lo que queramos
        console.log(dato2);
        document.getElementById('p1').innerText=`${dato2.name
} ${dato2.descrip}`;
      })
      .catch();
  });
});
```

```
}

window.addEventListener('load', ()=>{
  document.getElementById('addProduct')
              .addEventListener('submit', async
(event)=>{
    event.preventDefault();
    const nuevoProducto={
      id:"",
      name:document.getElementById('name').value,
      descrip:document.getElementById('descrip').
value
    };

    const dato2 = await
anadirProducto(nuevoProducto);

    //pintamos lo que queramos
    console.log(dato2);
    document.getElementById('p1')
              .innerText=`${dato2.name}
${dato2.descrip}`;
  });
});
```

post.js > ...

```
// POST con fetch
const SERVER = 'http://localhost:4000';
window.addEventListener('load', ()=>{
  document.getElementById('addProduct')
                .addEventListener('submit', (event)=>{
    event.preventDefault();
    const nuevoProducto={
      id:"",
      name:document.getElementById('name').value,
      descrip:document.getElementById('descrip').value
    };

    const promesa=fetch(SERVER+'/productos', {
      method: 'POST',
      body: JSON.stringify(nuevoProducto),
      headers: {
        'Content-Type': 'application/json'
      }
    })
    promesa
      .then((dato1)=>dato1.json())
      .then((dato2)=>{
        //pintamos lo que queramos
        console.log(dato2);
        document.getElementById('p1')
                .innerText=`${dato2.name} ${dato2.descrip}`;
      })
      .catch();
  });
});
```

post.js > ...

```
// POST con fetch y asunc await
const SERVER = 'http://localhost:4000';
async function anadirProducto(nuevoProducto){
  const dato1 = await fetch(SERVER+'/productos', {
    method: 'POST',
    body: JSON.stringify(nuevoProducto),
    headers: {
      'Content-Type': 'application/json'
    }
  });
  const dato2= await dato1.json();
  return dato2;
}

window.addEventListener('load', ()=>{
  document.getElementById('addProduct')
                .addEventListener('submit', async (event)=>{
    event.preventDefault();
    const nuevoProducto={
      id:"",
      name:document.getElementById('name').value,
      descrip:document.getElementById('descrip').value
    };

    const dato2 = await anadirProducto(nuevoProducto);

    //pintamos lo que queramos
    console.log(dato2);
    document.getElementById('p1')
                .innerText=`${dato2.name} ${dato2.descrip}`;
  });
});
```

Hacer varias peticiones simultáneamente. Promise.all

En ocasiones necesitamos hacer más de una petición al servidor.

Por ejemplo; para obtener los productos y sus categorías podríamos hacer:

```
function getTable(table) {
    return new Promise((resolve, reject) => {
      fetch(SERVER + table)
        .then(response => {
          if (!response.ok) {
            throw `Error ${response.status} de la BBDD: ${response.statusText}`
          }
          return response.json()
        })
        .then((data) => resolve(data))
        .catch((error) => reject(error))
    })
  }

  function getData() {
    getTable('/categories')
      .then((categories) => categories.forEach((category) => renderCategory(category)))
      .catch((error) => renderErrorMessage(error))
    getTable('/products')
      .then((products) => products.forEach((product) => renderProduct(product)))
      .catch((error) => renderErrorMessage(error))
  }
```

Pero si para renderizar los productos necesitamos tener las categorías, este código no nos lo garantiza ya que el servidor podría devolver antes los productos, aunque los hemos pedido después.

Una solución sería no pedir los productos hasta tener las categorías:

```
function getData() {
    getTable('/categories')
      .then((categories) => {
        categories.forEach((category) => renderCategory(category))
        getTable('/products')
          .then((products) => products.forEach((product) => renderProduct(product)))
          .catch((error) => renderErrorMessage(error))
      })
      .catch((error) => renderErrorMessage(error))
  }
```

pero esto hará más lento nuestro código al no hacer las 2 peticiones simultáneamente.

La solución es usar el método `Promise.all()` al que se le pasa un array de promesas a hacer y devuelve una promesa que:

- se resuelve en el momento en que todas las promesas se han resuelto satisfactoriamente o
- se rechaza en el momento en que alguna de las promesas es rechazada

El código anterior de forma correcta sería:

```
function getData() {
    Promise.all([
      getTable('/categories')
      getTable('/products')
    ])
    .then(([categories, products]) => {
      categories.forEach((category) => renderCategory(category))
      products.forEach((product) => renderProduct(product))
    })
    .catch((error) => renderErrorMessage(error))
  }
```

Lo mismo pasa si en vez de promesas usamos *async/await*. Si hacemos:

```
async function getTable(table) {
    const response = await fetch(SERVER + table)
    if (!response.ok) {
      throw `Error ${response.status} de la BBDD: ${response.statusText}`
    }
    const data = await response.json()
    return data
}

async function getData() {
  const responseCategories = await getTable('/categories');
  const responseProducts = await getTable('/products');
  categories.forEach((category) => renderCategory(category))
  products.forEach((product) => renderProduct(product))
}
```

tenemos el problema de que no comienza la petición de los productos hasta que se reciben las categorías. La solución con `Promise.all()` sería:

```
async function getData() {
    const [categories, products] = await Promise.all([
      getTable('/categories')
      getTable('/products')
    ])
    categories.forEach((category) => renderCategory(category))
    products.forEach((product) => renderProduct(product))
  }
```

Single Page Application

Ajax es la base para construir SPAs que permiten al usuario interactuar con una aplicación web como si se tratara de una aplicación de escritorio (sin “esperas” que dejen la página en blanco o no funcional mientras se recarga desde el servidor).

En una SPA sólo se carga la página de inicio (es la única página que existe) que se va modificando y cambiando sus datos como respuesta a la interacción del usuario.

Para obtener los nuevos datos se realizan peticiones al servidor (normalmente Ajax). La respuesta son datos (JSON, XML, …) que se muestran al usuario modificando mediante DOM la página mostrada (o podrían ser trozos de HTML que se cargan en determinadas partes de la página, o …).

Resumen de llamadas asíncronas

Una llamada Ajax es un tipo de llamada asíncrona que podemos hacer en Javascript.

Aunque hay otros tipos de llamadas asíncronas, como un `setTimeout()` o las funciones manejadoras de eventos.

Como hemos visto, para la gestión de las llamadas asíncronas tenemos varios métodos y los más comunes son:

- funciones *callback*
- *promesas*
- *async / await*

Cuando se produce una llamada asíncrona el orden de ejecución del código no es el que vemos en el programa, ya que el código de respuesta de la llamada no se ejecutará hasta completarse ésta.

Además, si hacemos varias llamadas tampoco sabemos en qué orden se ejecutarán sus respuestas, ya que depende de cuándo finalice cada una.

Si usamos funciones *callback* y necesitamos que cada función no se ejecute hasta que haya terminado la anterior, debemos llamarla en la respuesta a la función anterior, lo que provoca un tipo de código difícil de leer llamado *"callback hell"*.

Para evitar estos problemas surgieron las ***promesas*** que permiten evitar las funciones *callback* tan difíciles de leer. Y si necesitamos ejecutar secuencialmente las funciones evitaremos la pirámide de llamadas *callback* .

Aun así, el código no es muy claro. Para mejorarlo tenemos ***async* y *await*** . Estas funciones forman parte del estándar ES2017 por lo que no están soportadas por navegadores muy antiguos (aunque siempre podemos transpilar con *Babel*).

CORS

Cross-Origin Resource Sharing (CORS) es un mecanismo de seguridad que incluyen los navegadores y que por defecto impiden que se puedan realizar peticiones Ajax desde un navegador a un servidor con un dominio diferente al de la página cargada originalmente.

Si necesitamos hacer este tipo de peticiones necesitamos que el servidor al que hacemos la petición añada en su respuesta la cabecera *Access-Control-Allow-Origin* donde indiquemos el dominio desde el que se pueden hacer peticiones (o ***** para permitirlas desde cualquier dominio).

El navegador comprobará las cabeceras de respuesta y si el dominio indicado por ella coincide con el dominio desde el que se hizo la petición, ésta se permitirá.

Como en desarrollo normalmente no estamos en el dominio de producción (para el que se permitirán las peticiones) podemos instalar en el navegador la extensión *allow CORS* que al activarla deshabilita la seguridad CORS en el navegador.